Zipper-based Attribute Grammars and their Extensions*

Pedro Martins[1], João Paulo Fernandes[1,2], and João Saraiva[1]

[1] High-Assurance Software Laboratory (HASLAB/INESC TEC), Universidade do Minho, Portugal
[2] Reliable and Secure Computation Group ((rel)ease), Universidade da Beira Interior, Portugal
{prmartins,jpaulo,jas}@di.uminho.pt

Abstract. Attribute grammars are a suitable formalism to express complex software language analysis and manipulation algorithms, which rely on multiple traversals of the underlying syntax tree. Recently, Attribute Grammars have been extended with mechanisms such as references and high-order and circular attributes. Such extensions provide a powerful modular mechanism and allow the specification of complex fix-point computations. This paper defines an elegant and simple, zipper-based embedding of attribute grammars and their extensions as first class citizens. In this setting, language specifications are defined as a set of independent, off-the-shelf components that can easily be composed into a powerful, executable language processor. Several real examples of language specification and processing programs have been implemented in this setting.

1 Introduction

Attribute Grammars (AGs) [1] are a well-known and convenient formalism not only for specifying the semantic analysis phase of a compiler but also to model complex multiple traversal algorithms. Indeed, AGs have been used not only to specify real programming languages, like for example Haskell [2], but also to specify powerful pretty printing algorithms [3], deforestation techniques [4] and powerful type systems [5], for example.

All these attribute grammars specify complex and large algorithms that rely on multiple traversals over large tree-like data structures. To express these algorithms in regular programming languages is difficult because they rely in complex recursive patterns, and, most importantly, because there are dependencies between values computed in one traversal and used in following ones. In such cases, an explicit data structure has to be used to glue different traversal functions. In an imperative setting those values are stored in the tree nodes (which work as a

* This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within projects FCOMP-01-0124-FEDER-020532 and FCOMP-01-0124-FEDER-022701.

gluing data structure), while in a declarative setting such data structures have to be defined and constructed. In an AG setting, the programmer does not have to concern himself on scheduling traversals, nor on defining gluing data structures.

Recent research in attribute grammars is working in two main directions. Firstly, AG-based systems are supporting new extensions to the standard AG formalism that improve the AG expressiveness. Higher-order AGs (HOAGs) [6, 7] provide a modular extension to AGs. Reference AGs (RAGs) [8] allow the definition of references to remote parts of the tree, and, thus, extending the traditional tree-based algorithms to graphs. Finally, Circular AGs (CAGs) allow the definition of fix-point based algorithms. AG systems like Silver [9], JastAdd [10], and Kiama [11] all support such extensions. Secondly, attribute grammars are embedded in regular programming languages and AG fragments are first-class citizens: they can be analyzed, reused and compiled independently.

First class AGs provide: i) a full component-based approach to AGs where a language is specified/implemented as a set of reusable off-the-shelf components, and ii) semantic-based modularity, while traditional AG specifications use a (re-strict) syntax modular approach. Moreover, by using an embedding approach there is no need to construct a large AG (software) system to process, analyse and execute AG specifications: first class AGs reuse for free the mechanisms provided by the host language as much as possible, while increasing abstraction on the host language. Although this option may also entail some disadvantages, e.g. error messages relating to complex features of the host language instead of specificities of the embedded language, the fact is that an entire infrastructure, including libraries and language extensions, is readily available at a minimum cost. Also, the support and evolution of such infrastructure is not a concern.

This paper presents a novel technique combining these two AG advances.

First, we propose a concise embedding of AGs in `Haskell`. This embedding relies on the extremely simple mechanism of functional zippers. Zippers were originally conceived by Huet [12] to represent a tree together with a subtree that is the focus of attention, where that focus may move within the tree. By providing access to any element of a tree, zippers are very convenient in our setting: attributes may be defined by accessing other attributes in other nodes. Moreover, they do not rely on any advanced feature of `Haskell`. Thus, our embedding can be straightforwardly re-used in any other functional environment.

Second, we extend our embedding with the main AG extensions proposed to the AG formalism. In fact, we present the first embedding of HOAGs, RAGs and CAGs as first class attribute grammars. By this we are able to express powerful algorithms as the composition of AG reusable components. An approach that we have been using, e.g., in developing techniques for a language processor to implement bidirectional AG specifications and to construct a software portal.

2 Motivation

In this section we introduce the Desk language, that was proposed in [13], and that we will use as our running example throughout the paper. This language is

small enough to be completely defined here while still holding central characteristics of real programming languages, such as mandatory but unique declaration of all name entities that are used. The Desk language allows the definition of simple arithmetic expressions whose single operator is addition and that uses globally scoped variables. A concrete sentence in this language defines the sum of variables `x` and `y` with the value `1`, where `x` and `y` are set to `2` and `3`, respectively:

```
PRINT x + y + 1 WHERE x = 2, y = 3
```

Our goal here is similar to the one of [13]: we want to define a mapping from Desk sentences to assembly code for a simple machine with one register only. For the sentence above, we want to transform it into the following assembly program:

```
{ LOAD 2 (the value of x); ADD 3 (the value of y); ADD 1; PRINT 0; HALT 0 }
```

Implementing this transformation introduces typical language processing challenges such as lexical and syntactical analysis, name analysis through symbol table management, verification of static conditions, right-to-left processing and interpretation and code generation. Since declaration of entities may come after their usage, a traditional approach to solve this problem relies on complex, multiple traversal algorithms. In his original paper [13], Paaki proposed to implement this mapping using an AG with the following set of attributes:

```
code  - synthesized target code
name  - synthesized name of a constant
value - synthesized value of a constant or a number
ok    - synthesized attribute that indicates correcteness
envs  - synthesized environment (symbol table)
envi  - inherited environment (symbol table)
```

Attributes `envs` and `envi` both have the form of a list with `(name, value)` pairs representing a symbol table. Attribute `code` is the actual meaning of the grammar, i.e., the final result of processing a sentence, and it has the form of a list of pairs `(instruction, value)`. An important thing to notice is that an incorrect Desk phrase also yields a meaning. For example,

```
PRINT z WHERE x = 2, y = 3
```

produces the resulting code (i.e., has the following meaning):

```
{ HALT 0; PRINT 0; HALT 0 }
```

Next, we present the implementation of the Desk AG, as proposed in [13]:

```
(p1) Prog -> PRINT Exp Cons
     { Prog.code = if Cons.ok then Exp.code + (PRINT, 0) + (HALT, 0)
                              else (HALT,  0)
     , Exp.envi  = Cons.envs }
(p2) Exp₁ -> Exp₂ '+' Fact
     { Exp₁.code = if Fact.ok then Exp₂.code + (ADD, Fact.value)
                              else (HALT, 0)
     , Exp₂.envi  = Exp₁.envi
     , Fact.envi  = Exp₁.envi }
(p3) Exp -> Fact
     { Exp.code   = if Fact.ok then (LOAD, Fact.value) else (HALT, 0)
```

```
     , Fact.envi  = Exp.envi }
(p4) Fact -> Name
     { Fact.ok    = isin (Name.name, Fact.envi)
     , Fact.value = getvalue (Name.name, Fact.envi) }
(p5) Fact -> Number
     { Fact.ok = true, Fact.value = Number.value }
(p6) Name -> Id
     { Name.name = Id.name }
(p7) Cons -> empty
     { Cons.ok = true, Cons.envs = () }
(p8) Cons -> WHERE DefList
     { Cons.ok = DefList.ok, Cons.envs = DefList.envs }
(p9) DefList1 -> DefList2 ',' Def
     { DefList1.ok = DefList2.ok and not isin (Def.name, DefList2.envs)
     , DefList1.envs = DefList2.envs + (Def.name, Def.value) }
(p10) DefList -> Def
     { DefList.ok = true, DefList.envs = (Def.name, Def.value) }
(p11) Def -> Name '=' Number
     { Def.name = Name.name, Def.value = Name.value }
```

A definition `(p n) production {semantic rules}` is used to associate concrete semantics (using `semantic rules` to define attribute values) to the syntax (defined by context-free grammar `productions`) of a language. In a production, when the same non-terminal symbol occurs more than once, each occurrence is denoted by a subscript (starting from `1` and counting left to right).

In this particular case, it is assumed that the values of attributes `name` and `value` are externally provided, e.g., by a lexical analyzer. Also, we use constructions `if then else`, `and` and `not` assuming their standard interpretation; `+` is used for list consing, `isin` to check wether a value is contained within a symbol table and `getvalue` to extract a value from a symbol table, returning `0` if it does not exist[3]. Conventional constant functions are also used, such as the integer `0`, the Boolean `true` and the empty list `()`.

3 Zipper-based Attribute Grammars

In this section we show how we can implement Desk as an AG embedded in Haskell relying on the concept of functional zippers, that we start by revising.

3.1 Functional Zippers

In our work we have used the generic zipper library of [14]. It works for both homogeneous and heterogeneous datatypes, and data-types for which an instance of the `Data` and `Typeable` type classes [15] are available can be traversed.

In order to illustrate how we may use zippers, we consider the following Haskell data-type straightforwardly obtain from the syntax of the Desk language.

[3] The traditional definition of AGs only permits semantic rules of the form `X.a = f(...)`, forcing the use of identity functions for constants. For clarity and simplicity, we allow their direct usage in attribute definitions.

```
data Root    = Root  Prog
data Prog    = PRINT Exp Cons
data Exp     = Add   Exp Fact    | Fact Fact
data Fact    = Name  Name        | Number String
data Name    = Id Constant
data Cons    = EmptyCons         | WHERE DefList
data DefList = Comma DefList Def | Def Def
data Def     = Equal Name Value
type Constant    = String
type Value       = Int
type SymbolTable = [(String,String)]
```

We may use this data-type to represent `PRINT x + y + 1 WHERE x = 2, y = 3` as the following `program`:

```
exp     = Add (Add (Fact (Name (Id "x")))
                   (Name (Id "y")))
              (Number 1)
deflst  = WHERE (Comma (Def (Equal (Id "y") 5))
                       (Equal (Id "x") 3))
program = PRINT exp deflst
```

In order to navigate on `program`, we start by wrapping it up using the library-provided function `toZipper :: Data a => a -> Zipper a`:

```
program' = toZipper (Root program)
```

We end up with an aggregate data structure which is easy to traverse and update. For example, we may move the focus of attention on `program'` from the topmost node to the `exp` node as follows:[4]

```
exp' = (getHole . down . down) program'
```

The library function `down` goes down to the leftmost (immediate) child of a node whereas function `getHole` extracts the node under focus from a zipper.

3.2 Desk as an Embedded Attribute Grammar

On top of the zipper library of [14], we have implemented several simple combinators that facilitate the embedding of attribute grammars. In particular, we have defined: `(.$) :: Zipper a -> Int -> Zipper a` for accessing any child of a structure given by its index starting in `1`; `parent :: Zipper a -> Zipper a` to move the focus to the parent of a concrete node, and, to check whether the current location is a sibling of a tree node, `(.|) :: Zipper a -> Int -> Bool`.

We may now define each attribute of the Desk attribute grammar. For synthesizing the name of a constant, as defined in the semantics of production `(p6)` and (part of) production `(p11)` we define an attribute `name` as follows[5].

[4] For totality, the results of functions `down :: Zipper a -> Maybe (Zipper a)` and `getHole :: Zipper a -> Maybe b` are within `Maybe`. As the analysis of their results is provided by our combinators, we simplify the example by abstracting this analysis.

[5] Function `constructor` exposes the type of the node under focus, and function `lexeme` simulates a standard lexer, and both can be automatically generated.

```
name :: Zipper Root -> String
name ag = case (constructor ag) of "Id"    -> lexeme ag
                                   "Equal" -> name (ag.$1)
```

where `Zipper Root` is the type of an instance of `Root` embedded inside the `Zipper`.

The purpose of the attributes `envs` and `envi` is, respectively, to compute and to appropriately pass around an environment mapping constant names to values in accordance with the bindings in `DefList`. For a node being a `Prog`, the inherited environment `envi` is given by the synthesized environment `envs` of its `Cons` child, as defined in `(p1)`. For any other node, `envi` is accessed in its parent node:

```
envi :: Zipper Root -> SymbolTable
envi ag = case (constructor ag) of "PRINT"   -> envs (ag.$2)
                                   otherwise -> envi (parent ag)
```

Attribute `envi` is copied from the root node where it is computed to any other node. There is usually a primitive that allows the programmer to define this type of attributes without having to specify them throughout the grammar: in Silver [9], for example, this is called `autocopy`. Our solution relies on Haskell 's `case/otherwise` construction to implement a similar feature.

The synthesized environment `envs` goes down a sentence in search for the constants defined in it. When one such definition is found, a pair `(c, v)`, where `c` is the constant `name` and `v` the `value` being set for it, is added to the environment:

```
envs :: Zipper Root -> SymbolTable
envs ag = case (constructor ag) of
          "EmptyCons" -> []
          "WHERE" -> envs (ag.$1)
          "Comma" -> (name (ag.$2), value (ag.$2)) : envs (ag.$1)
          "Def"   -> [(name (ag.$1), value (ag.$1))]
```

The value of a constant or a number is given by attribute `value` as follows:

```
value :: Zipper Root -> String
value ag = case (constructor ag) of
           "Name"   -> getValue (name (ag.$1)) (envi ag)
           "Number" -> lexeme ag
           "Equal"  -> lexeme ag
```

The value of a constant `Name c` occurring in the `Expression` part of a sentence must be searched for in the environment, which is precisely what `getValue` does. The value of `Number v` or the constant definition `Equal c v` is simply `v`.

The attribute `ok` checks if a variable is defined once and only once:

```
ok :: Zipper Root -> Bool
ok ag = case (constructor ag) of
        "Name"   -> isIn (name (ag.$1)) (envi ag)
        "Number" -> True
        "EmptyCons" -> True
        "WHERE"  -> ok (ag.$1)
        "Comma"  -> ok (ag.$1) && not isIn (name (ag.$2)) (envs (ag.$1))
        "Def"    -> True
```

The synthesized attribute `code` reuses the defined attributes to generate code.

```
code :: Zipper Root -> String
code ag = case (constructor ag) of
          "Root"  -> code (ag.$1)
          "PRINT" -> if ok (ag.$2)
                     then code (ag.$1) ++ "PRINT, 0" ++ "HALT, 0"
                     else "HALT, 0"
          "Add"   -> if (ok (ag.$2))
                     then code (ag.$1) ++ "ADD, " ++ value (ag.$2)
                     else "HALT, 0"
          "Fact"  -> if (ok (ag.$1))
                     then  "LOAD, " ++ value (ag.$1)
                     else "HALT, 0"
```

In this section, we have embedded the Desk analysis as an AG in Haskell. Our solution is simple and elegant, easy to implement, to analyze and to extend.

A difference between our embedding and the traditional definition of AGs is that in the former, an attribute is defined as a semantic function on tree nodes, while in the latter the programmer defines on one production exactly how many and how attributes are computed. Nevertheless, we argue that this difference does not impose increasing implementation costs as the main advantages of the attribute grammar setting still hold: attributes are modular, their implementation can be sectioned by sites in the tree and as we will see inter-attribute definitions work exactly the same way. What is more, our embedding might provide an easier setting for debugging as the entire definition of one attribute is localized in one semantic function. Furthermore, we believe that the individual attribute definitions in our embedding can straightforwardly be understood and derived from their traditional definition on an attribute grammar system, as can be observed comparing the attribute definitions in the previous section with the ones in this section.

A traditional advantage of the embedding of domain-specific languages in a host language is the use of target language features as native. In our case, this applies, e.g., to the Haskell functions `&&` for Boolean conjunction, `not` for Boolean negation and `++` for list concatenation, whereas on specific AG systems the set of functions is usually limited and pre-defined. Also, regarding distribution of language features for dynamical load and separate compilation, it is posible to divide an AG in modules that, e.g., may contain contain data types (representing the grammar) and functions (representing the attributes).

4 Zipper-based Attribute Grammar Extensions

After showing first class attribute grammars embedded in a zipper framework, we present the embedding of three well known AG extensions.

4.1 Referenced Attribute Grammars

Referenced Attribute Grammars [8] allow references to arbitrary nodes in the tree, and attributes attached to those nodes to be accessed via the referenced

attributes. Because RAGs allow nodes to reference any node in the tree (not only their children), they allow the expression of graph-based algorithms.

In the original Desk AG, the inherited attribute `envi` is used to collect and pass context information to the expression part of a sentence. However, if this language evolves to allow, e.g., type definitions, then a complete re-write of the symbol table with the respective attributes and semantics may be needed. By using RAGs, the symbol table is promoted to contain references to locations in the tree. As a result, if the definition part evolves, then the attribute references still point to the evolved tree, and changes are much easier to carry.

In our embedding, references are represented by zippers whose focus points to relevant tree locations. This implies changes on the symbol table's data type, its construction and the lookup semantic function that uses it:

```
type SymbolTable = [(String, Zipper Root)]

envs :: Zipper Root -> SymbolTable
envs ag = case (constructor ag) of
          "EmptyCons" -> []
          "WHERE" -> envs (ag.$1)
          "Comma" -> envs (ag.$1) ++ [( name (ag.$2), ag.$2 )]
          "Def"   -> [( name (ag.$1), ag.$1 )]

isIn :: String -> SymbolTable -> Bool
isIn _ [] = False
isIn name ((a,b):xs) = if (name == a) then True else isIn name xs

getValue :: String -> SymbolTable -> Bool
getValue name ((a,b):xs) = if (name == a) then (value b)
                                  else (getValue name xs)
```

This definition is very similar to the AG in Section 3, with the main difference being the fact that, since the symbol table is composed by references, the semantic function `getValue` has to use the attribute `value` to extract the actual assigned values, where it only had to return information contained in the list. This is the general approach in RAGs.

4.2 Higher-Order Attribute Grammars

Higher-order attribute grammars [6] are an important extension of AGs because they allow both tree changes during attribute evaluation, and the definition of any (first-order) recursive functions as AG computations. Moreover, they also provide a component-based (modular) approach to AG specifications [7].

In our running example the functions `getValue` and `isIn` are semantically expressed, contrary to the AG, while on a HOAG setting those computations are promoted to higher-order attributes.

We start by creating a new data type for the symbol table:

```
data Root_ho = Root_ho SymbolTable
data SymbolTable = NilST | ConsST Tuple SymbolTable
type Tuple = (String, String)
```

The symbol table becomes a tree-based structure with clear constructors and names for tree nodes. Having defined these data types, we only need to express the lookup operations as attribute computations.

```
isIn :: String -> Zipper Root_ho -> Bool
isIn name ag = case (constructor_ho ag) of
            "Root_ho" -> isIn name (ag.$1)
            "NilST"   -> False
            "ConsST"  -> isIn name (ag.$1) || isIn name (ag.$2)
            "Tuple"   -> lexeme_ho z == name

getValue :: String -> Zipper Root_ho -> String
getValue name ag = case (constructor_ho ag) of
              "Root_ho" -> getValue (ag.$1)
              "ConsST" -> if (lexeme_ho (ag.$1) == name)
                          then (lexeme_ho (ag.$1))
                          else (getValue name (ag.$2))
```

Having modelled the two lookup functions, we now need to focus on the part of the specification where those functions are called. Instead of a function call, in a HOAG setting we need to instantiate the higher-order tree as a zipper, as shown next (we include the relevant productions only):

```
value :: Zipper Root -> String
value ag = case (constructor ag) of
          "Name" -> getValue (name ag.$1) (toZipper (Root_ho (envi ag)))

ok :: Zipper Root -> String
ok ag = case (constructor ag) of
        "Name"  -> isIn (name ag.$1) (toZipper (Root_ho (envi ag)))
        "Comma" -> (ok ag.$1) && not
                   isIn (name ag.$2) (toZipper (Root_ho (envs ag.$1)))
```

Like in standard HOAG specifications, as supported by LRC [16], a call to a semantic function (in a classical AG) is transformed into a higher-order tree/attribute. In our embedding the function `toZipper` is used to model this.

4.3 Circular Attribute Grammars

HOAGs allow expressing first-order computations but several algorithms, such as type inference, rely on fix-point computations. In order to express these algorithms in a AG setting, we need to consider Circular Attribute Grammars [17].

As an example, lets imagine the revised version of Desk as considered by [13], where assignments can be symbolical and their order is not relevant:

```
PRINT x + y + 1 WHERE x = y, z = 1, y = z
```

To process this Desk expression, we need a fixed-point evaluation strategy. The general idea is to start with a bottom value, $\perp$, and compute approximations

of the final result until it is not changed anymore, that is, the least fixed point is reached: `x = ⊥; x = f(x); x = f(f(x)); ....`

To guarantee the termination of this computation, it must be possible to test the equality of the result (with ⊥ being its smallest value). With this, the sequence `x = ⊥; x = f(x); x = f(f(x)); ...` will return the final result, in the form `f(f(...f(⊥)...))`.

Of course, this solution might produce an infinite loop in cases such as:

```
PRINT x + y + 1 WHERE x = y, y = x
```

While this is undesired, this assignment is actually impossible to solve (besidesm it corresponds to an invalid Desk phrase).

Next, we present the Haskell function that implements this definition:

```
fixed-point :: Eq a => (a -> a) -> a -> a
fixed-point f s | s == next = s
                | otherwise = fixed-point f next
                where next = f s
```

This is a standard Haskell solution, that takes as argument `f`, an input `s` and applies the function indefinitely until it can not perform more changes to the input, i.e., until `f(s) == s`. It is easy to imagine in Desk a call such as `fixed-point solver symbol-table`, where `solver` solves as much assignments as possible in one traverse, and is applied until no more assignments can be resolved. Such improvement to Desk would successfully update the original implementation of the language to solve a new class of circular dependencies. Despite successful, this solution is not preferable since it forces standard semantic approaches and we loose part of the expressive power of AGs. Therefore, our approach is to define a new attribute, `isSolved`, that terminates the fixed-point computation. This is a more desirable way of controlling the fixed-point process since we are not constrained to function equality and we can do so in an AG fashion, by modularly creating definitions per tree node, as shown next.

```
isSolved :: Zipper Root_ho -> Bool
isSolved ag = case (constructor_ho ag) of
              "Root_ho" -> auxIsSolved (ag.$1)
              otherwise -> isSolved (parent ag)

auxIsSolved :: Zipper Root_ho -> Bool
auxIsSolved ag = case (constructor_ho ag) of
                 "Root_ho"     -> auxIsSolved (ag.$1)
                 "ConsST"      -> (auxIsSolved ag.$1) &&
                                  (auxIsSolved ag.$2)
                 "NilST"       -> True
                 "TupleInt"    -> True
                 "TupleString" -> False
```

The attribute `isSolved` exists only to ensure that this test is performed on the whole HOAG, and not only on a subpart of it. Therefore, it goes all the way to the top where it calls another attribute, `auxIsSolved`. The attribute `auxIsSolved`

goes through the tree and checks if any of the position contains an assignment to another variable. If it does, the symbol table is not "complete", i.e., there are assignments still left to solve.

Secondly, to make the example more interesting, we shall implement this using a high-order AG. HOAG is being constantly calculated until a certain condition is met (remember, in the traditional fixed-point approach, this condition would be that two subsequent computations produce the same output). It is a good idea to use an HOAG because, as we have seen in Section 4.2, this type of grammars are much more easier to handle, to manage and to reason about.

Next, we present the attributes responsible for solving as much assignments as possible of the high-order symbol table in one traverse:

```
solve :: Zipper Root_ho -> Zipper Root_ho
solve ag = case (constructor_ho ag) of
           "Root_ho" -> solve (ag.$1)
           "NilST"   -> NilST
           "ConsST"  -> ConsST (check ag.$1) (solve ag.$2)

check :: Zipper Root_ho -> Bool
check ag = case (constructor_ho ag) of
           "TupleInt"    -> lexeme_ho ag
           "TupleString" -> substitute (solvedSymbols ag)
                                       (lexeme_ho ag)
```

The attribute `solve` goes recursively through the tree and calls the attribute `check` on every tree node. If this node contains an assignment to a number, `check` does not do anything. On the other hand, if the assignment is to a variable, `check` uses a supporting semantic function, `substitute`, that takes as argument the unresolved assignment and a list of all the resolved assignments that exist in the symbol table and sees if any of this information can be used. The attribute that creates the list of resolved assignments is presented next.

```
solvedSymbols :: Zipper Root_ho -> [(String, Int)]
solvedSymbols ag = case (constructor_ho ag) of
                   "Root_ho" -> auxSolvedSymbols (ag.$1)
                   otherwise -> solvedSymbols (parent ag)

auxSolvedSymbols :: Zipper Root_ho -> [(String, Int)]
auxSolvedSymbols ag = case (constructor_ho ag) of
                      "ConsST"      -> auxSolvedSymbols (ag.$1) ++
                                       auxSolvedSymbols (ag.$2)
                      "NilST"       -> []
                      "TupleInt"    -> [(lexeme z, lexeme z)]
                      "TupleString" -> []
```

With all the necessary attributes implemented, we only have to define the attribute that applies this fixed-point strategy on the HOAG. As stated earlier, the idea of this fixed-point computation is to indefinitely apply a computation (`solve`) until a stop condition is reached (`isSolved`).

```
fixed-point ag = case (constructor_ho ag) of
           "Root_ho" -> if (isSolved ag) then ag
                        else fixed-point (toZipper (Root_ho (solve ag.$1)))
           otherwise  -> fixed-point (parent ag)
```

This way, we solved the cyclic dependencies imposed by a new version of Desk without loosing the modularity and expressiveness of AGs in our embedding. What is left to do is to call this `fixed-point` attribute immediately after the symbol table is created, namely in the original attribute `envi`.

5 Related Work

In this paper, we have proposed a zippers-based embedding of attribute grammars in a functional language. The implementations we obtain are modular and do not rely on laziness. We believe that our approach is the first that deals with arbitrary tree structures while being applicable in both lazy and strict settings. Furthermore, we have been able to implement in our environment all the standard examples that have been proposed in the attribute grammar literature. This is the case of `repmin` [18], `HTML table formatting` [7], and `smart parentesis`, an illustrative example of [9], that are available from the first author's webpage and that we will include in an extended version of the paper.

Moreover, the navigation via a generic zipper that we envison here has applications in other domains: *i*) our setting is being used to create combinator languages for process management [19] which themselves are fundamental to a platform for open source software analysis and certification [20, 21]; and *ii*), the setting that we propose was applied on a prototype for bidirectional transformations applied to programming environments for scientific computing.

Below we survey only works most closely related to ours: works in the realm of functional languages and attribute grammar embeddings.

Zipper-based approaches. Uustalu and Vene have shown how to embed attribute computations using comonadic structures, where each tree node is paired with its attribute values [22]. This approach is notable for its use of a zipper as in our work. However, it appears that this zipper is not generic and must be instantiated for each tree structure. Laziness is used to avoid static scheduling. Moreover, their example is restricted to a grammar with a single non-terminal and extension to arbitrary grammars is speculative.

Badouel *et al.* define attribute evaluation as zipper transformers [23]. While their encoding is simpler than that of Uustalu and Vene, they also use laziness as a key aspect and the zipper representation is similarly not generic. This is also the case of [24], that also requires laziness and forces the programmer to be aware of a cyclic representation of zippers.

Yakushev *et al.* describe a fixed point method for defining operations on mutually recursive data types, with which they build a generic zipper [25]. Their approach is to translate data structures into a generic representation on which traversals and updates can be performed, then to translate back. Even though their zipper is generic, the implementation is more complex than ours and incurs

the extra overhead of translation. It also uses more advanced features of Haskell such as type families and rank-2 types.

Non-zipper-based approaches. Circular programs have been used in the past to straightforwardly implement AGs in a lazy functional language [26, 27]. These works, in contrast to our own, rely on the target language to be lazy, and their goal is not to embed AGs: instead they show that there exists a direct correspondence between an attribute grammar and a circular program.

Regarding other notable embeddings of AGs in functional languages [28–30], they do not offer the modern AG extensions that we provided, with the exception of [30] that uses macros to allow the definition of higher-order attributes. Also, these embeddings are not based on zippers, rely on laziness and use extensible records [28] or heterogeneous collections [29, 30]. The use of heterogeneous lists in the second of these approaches replaces the use in the first approach of extensible records, which are no longer supported by the main Haskell compilers. In our framework, attributes do not need to be collected in a data structure at all: they are regular functions upon which correctness checks are statically performed by the compiler. The result is a simpler and more modular embedding. On the other hand, the use of these data structures ensures that an attribute is computed only once, being then updated to a data structure and later found there when necessary. In order to guarantee such a claim in our setting we need to rely on memoization strategies, often costly in terms of performance.

Our embedding does not require the programmer to explicitly combine different attributes nor does it require combination of the semantic rules for a particular node in the syntax tree, as is the case in the work of Viera *et al.* [29, 30]. In this sense, our implementation requires less effort from the programmer.

The Kiama library embeds attribute grammars in Scala [11]. This embedding is not purely functional, but uses generic 'parent' and similar operations to access the structure, instead of having more traditional inherited attribute definitions.

In general, when designing a Domain Specific Language (DSL), it is often the case that "syntax is not quite right" [31]. With this observation, the author claims that DSLs must be as close to the language being embedded as possible. Our DSL for AGs closely resembles custom AG languages, so we have the notational power without incurring the implementation cost of a custom language.

6 Conclusions and Future Work

In this paper we have presented the first embedding of modern AG extensions using a concise and elegant zipper-based implementation. We have presented how reference, higher-order and circular attribute grammars can be expressed as first class AGs in this setting. As a result, complex multiple traversal algorithms can be expressed in this setting in an off-the-shelf set of reusable components.

We have presented our embedding in the Haskell programming language, despite not relying on any advanced feature of Haskell (namely on lazy evaluation). Thus, similar concise embeddings can be defined in other declarative languages.

As we have shown both by the example presented and by the ones available online, our simple embedding provides the same expressiveness of modern, large and more complex attribute grammar based systems.

As part of our future research, we plan to: i) improve attribute definition by referencing non-terminals instead of (numeric) positions on the right-hand side of productions; and ii) wherever possible, benchmark our embedding against other AG embeddings and systems.

References

1. Knuth, D.: Semantics of Context-free Languages. Mathematical Systems Theory **2**(2) (June 1968) *Correction: Mathematical Systems Theory* 5 (1), March 1971.
2. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the utrecht haskell compiler. In Weirich, S., ed.: Haskell, ACM (2009) 93–104
3. Swierstra, D., Azero, P., Saraiva, J.: Designing and Implementing Combinator Languages. In Swierstra, D., Henriques, P., Oliveira, J., eds.: 3rd Summer School on Adv. Funct. Programming. Volume 1608 of LNCS Tutorial. (1999) 150–206
4. Fernandes, J.P., Saraiva, J.: Tools and Libraries to Model and Manipulate Circular Programs. In: PEPM'07: Proceedings of the ACM SIGPLAN 2007 Symposium on Partial Evaluation and Program Manipulation, ACM Press (2007) 102–111
5. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Iterative type inference with attribute grammars. In Visser, E., Järvi, J., eds.: GPCE, ACM (2010) 43–52
6. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. SIGPLAN Not. **24**(7) (June 1989) 131–145
7. Saraiva, J., Swierstra, S.D.: Generating spreadsheet-like tools from strong attribute grammars. In Pfenning, F., Smaragdakis, Y., eds.: GPCE. Volume 2830 of LNCS., Springer (2003) 307–323
8. Magnusson, E., Hedin, G.: Circular reference attributed grammars - their evaluation and applications. Sci. Comput. Program. **68**(1) (2007) 21–37
9. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Electron. Notes Theor. Comput. Sci. **203**(2) (2008) 103–116
10. Ekman, T., Hedin, G.: The jastadd extensible java compiler. SIGPLAN Not. **42**(10) (October 2007) 1–18
11. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. Electron. Notes Theor. Comput. Sci. **253**(7) (2010) 205–219
12. Huet, G.: The zipper. Journal of Functional Programming **7**(5) (1997) 549–554
13. Paakki, J.: Attribute grammar paradigms a high-level methodology in language implementation. ACM Comput. Surv. **27**(2) (June 1995) 196–255
14. Adams, M.D.: Scrap your zippers: a generic zipper for heterogeneous types. In: Proceedings of the 6th ACM SIGPLAN workshop on Generic programming. WGP '10, New York, NY, USA, ACM (2010) 13–24
15. Lämmel, R., Jones, S.P.: Scrap your boilerplate: a practical design pattern for generic programming. In: Procs. of the 2003 ACM SIGPLAN Inter. WorkShop on Types in Language Design and Implementation. (TLDI '03), ACM (2003) 26–37
16. Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In Koskimies, K., ed.: 7th International Conference on Compiler Construction. Volume 1383 of LNCS., Springer-Verlag (1998) 298–301
17. Magnusson, E., Hedin, G.: Circular reference attributed grammars - their evaluation and applications. Sci. Comput. Program. **68**(1) (August 2007) 21–37

18. Bird, R.: Using circular programs to eliminate multiple traversals of data. Acta Informatica **21** (1984) 239–250
19. Martins, P., Fernandes, J.P., Saraiva, J.: A purely functional combinator language for software quality assessment. In: Symposium on Languages, Applications and Technologies (SLATE '12). Volume 21 of OASICS., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012) 51–69
20. Martins, P., Fernandes, J.P., Saraiva, J.: A web portal for the certification of open source software. In: 6th International Workshop on Foundations and Techniques for Open Source Software Certification (OPENCERT '12) (to appear), LNCS (2012)
21. Martins, P., Carvalho, N., Fernandes, J.P., Almeida, J.J., Saraiva, J.: A framework for modular and customizable software analysis. In: 13th Int. Conf. on Computational Science and Its Applications (ICCSA 2013). LNCS (7972) (2012) 443–458
22. Uustalu, T., Vene, V.: Comonadic functional attribute evaluation. Trends in Functional Programming, Intellect Books (10) (2005) 145–162
23. Badouel, E., Fotsing, B., Tchougong, R.: Yet another implementation of attribute evaluation. Research Report RR-6315, INRIA (2007)
24. Badouel, E., Fotsing, B., Tchougong, R.: Attribute grammars as recursion schemes over cyclic representations of zippers. Electronic Notes Theory Computer Science **229**(5) (2011) 39–56
25. Yakushev, A.R., Holdermans, S., Löh, A., Jeuring, J.: Generic programming with fixed points for mutually recursive datatypes. In: Procs. of the 14th ACM SIGPLAN International Conference on Functional programming. (2009) 233–244
26. Johnsson, T.: Attribute grammars as a functional programming paradigm. In: Functional Programming Languages and Computer Architecture. (1987)
27. Kuiper, M., Swierstra, D.: Using attribute grammars to derive efficient functional programs. In: Computing Science in the Netherlands. (November 1987)
28. de Moor, O., Backhouse, K., Swierstra, S.D.: First-class attribute grammars. Informatica (Slovenia) **24**(3) (2000)
29. Viera, M., Swierstra, D., Swierstra, W.: Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell. In: Procs. of the 14th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'09). (2009) 245–256
30. Viera, M.: First Class Syntax, Semantics, and Their Composition. PhD thesis, Utrecht University, The Netherlands (2013)
31. Siek, J.: General purpose languages should be metalanguages. In: Procs. of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. (2010) 3–4